Functional programming
Lecture 01 — First steps

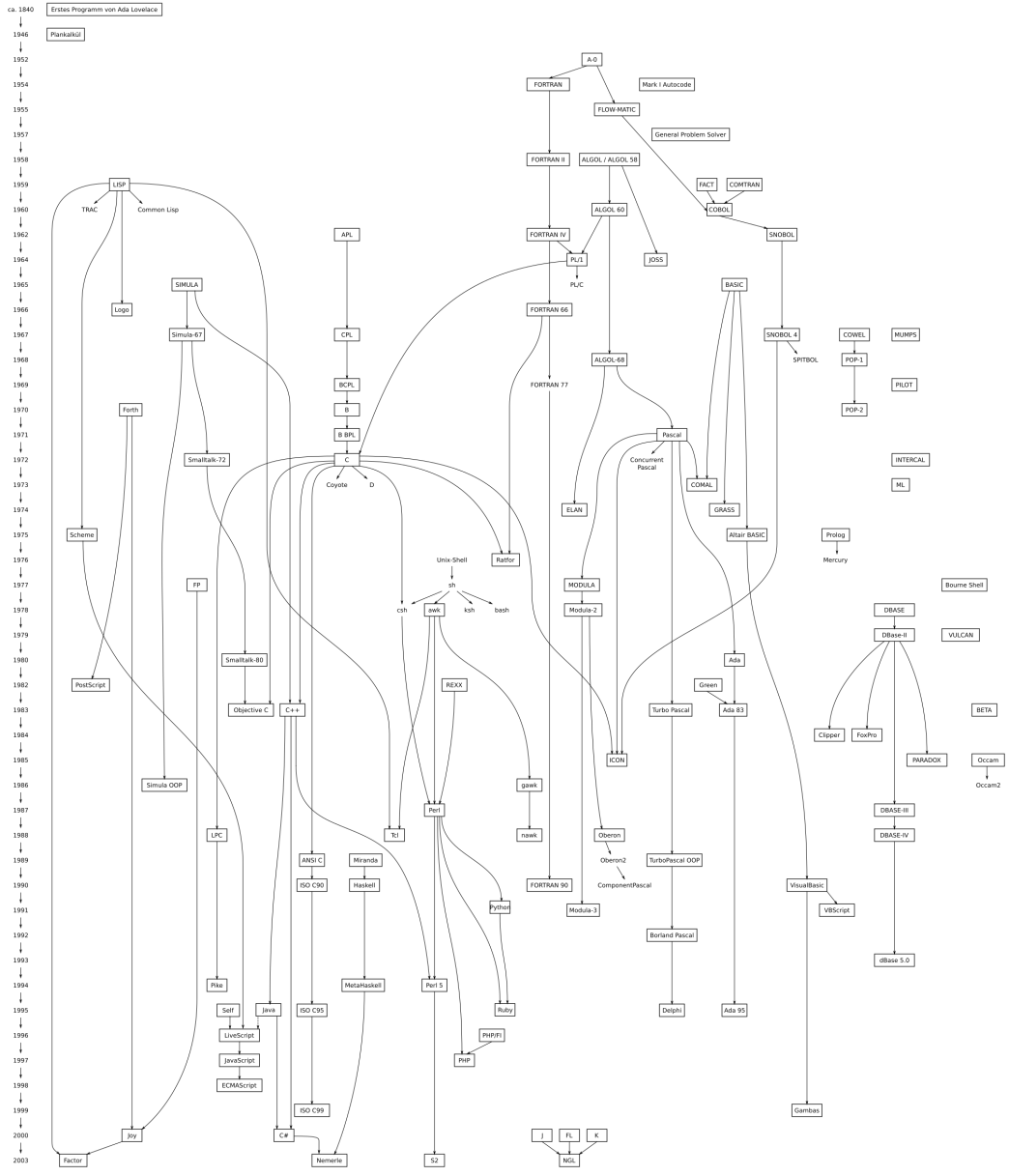
Stéphane Vialette
stephane.vialette@univ-eiffel.fr

November 6, 2023

Laboratoire d'Informatique Gaspard-Monge, UMR CNRS 8049,
Université Gustave Eiffel

Functional programming concepts

Genealogy of programming languages



Functional languages

Main functional programming languages

Lisp

Lisp (historically, LISP) is a family of computer programming languages with a long history and a distinctive, fully parenthesized prefix notation. Originally specified in 1958, Lisp is the second-oldest high-level programming language in widespread use today. (Only Fortran is older, by one year.)

Main functional programming languages

Erlang

Erlang is a general-purpose, concurrent, functional programming language, as well as a garbage-collected runtime system.

Main functional programming languages

Elixir

Elixir is a functional, concurrent, general-purpose programming language that runs on the Erlang virtual machine (BEAM).

Main functional programming languages

F#

F# is a strongly typed, multi-paradigm programming language that encompasses functional, imperative, and object-oriented programming methods. It is being developed at Microsoft Developer Division and is being distributed as a fully supported language in the .NET framework.

Main functional programming languages

Ocaml

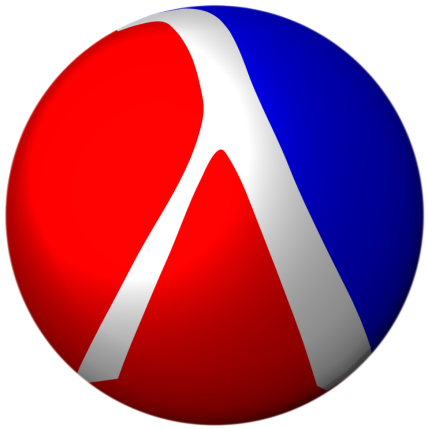
Ocaml, originally named Objective Caml, is the main implementation of the programming language Caml. OCaml's toolset includes an interactive top-level interpreter, a bytecode compiler, a reversible debugger, a package manager (OPAM), and an optimizing native code compiler.

Main functional programming languages

Clojure

Clojure is a dialect of the Lisp programming language. Clojure is a general-purpose programming language with an emphasis on functional programming. It runs on the Java virtual machine and the Common Language Runtime.

Main functional programming languages



Racket

Racket, formerly PLT Scheme, is a general purpose, multi-paradigm programming language in the Lisp-Scheme family. One of its design goals is to serve as a platform for language creation, design, and implementation

Main functional programming languages

Elm

Elm is a domain-specific programming language for declaratively creating web browser-based graphical user interfaces. Elm is purely functional, and is developed with emphasis on usability, performance, and robustness.

Main functional programming languages

Scala

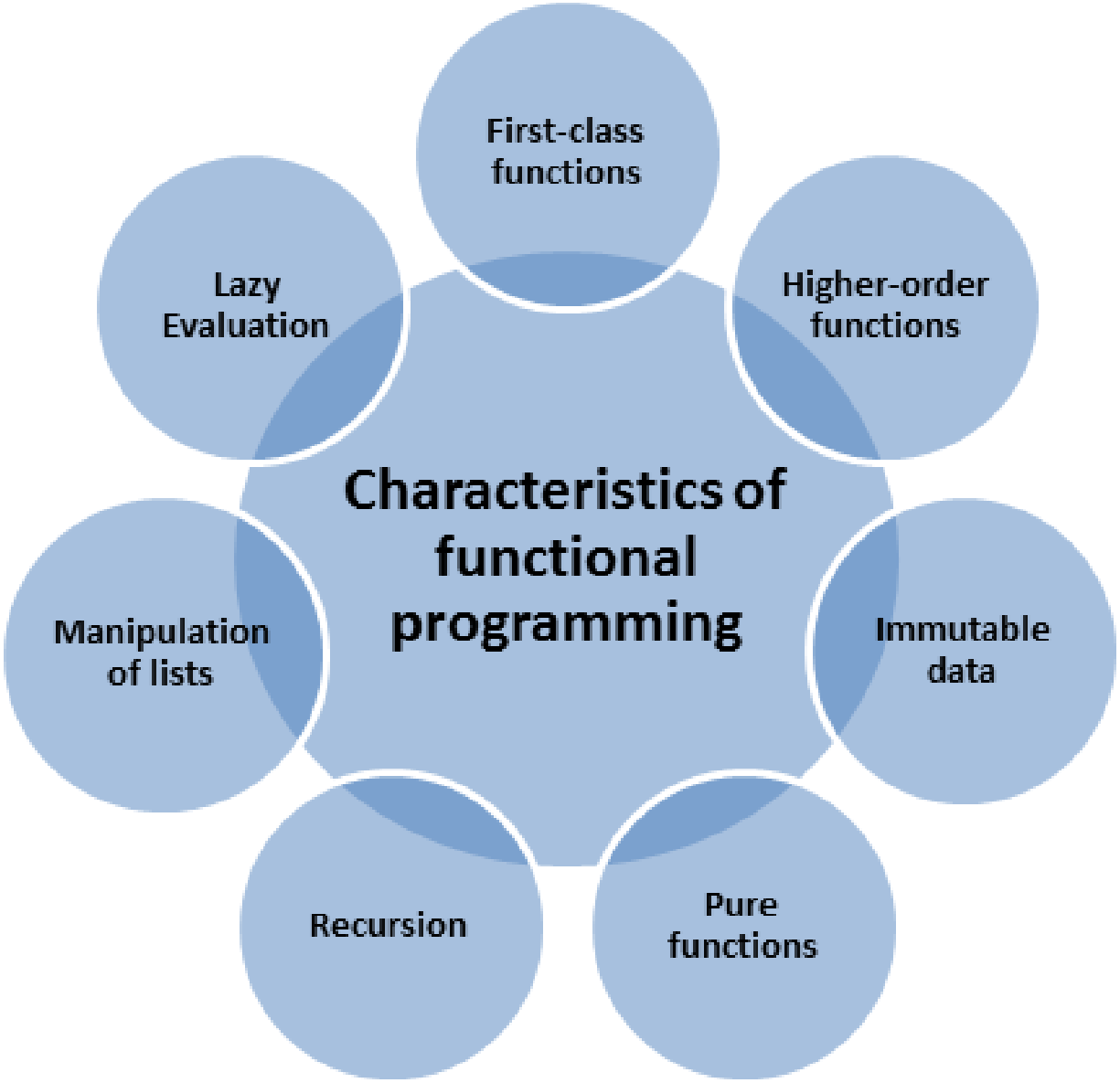
Scala is a general-purpose programming language providing support for functional programming and a strong static type system. Designed to be concise, many of Scala's design decisions aimed to address criticisms of Java.

Main functional programming languages

Haskell

Haskell is a general-purpose, statically-typed, purely functional programming language with type inference and lazy evaluation. Designed for teaching, research and industrial applications, Haskell has pioneered a number of programming language features such as type classes, which enable type-safe operator overloading, and monadic IO. Haskell's main implementation is the Glasgow Haskell Compiler (GHC). It is named after logician Haskell Curry.

Characteristics of functional programming



Haskell

- Haskell is a compiled, statically typed, functional programming language.
- It was created in the early 1990s as one of the first open-source purely functional programming languages.
- It is named after the American logician Haskell Brooks Curry.

Glasgow Haskell Compiler

- Concise programs
- Powerful type system
- List comprehensions
- Recursive functions
- High-order functions
- Effectful functions
- Generic functions
- Lazy evaluation
- Equational reasoning

Haskell landscape

The imperatives

- GHC: state-of-the-art, open source, compiler and interactive environment for the functional language Haskell.
- GHCi: GHC's interactive environment.
- Hackage: Haskell community's central package archive of open source software.

Haskell landscape

Testing Frameworks

- QuickCheck: powerful testing framework where test cases are generated according to specific properties.
- HUnit: unit testing framework similar to JUnit.
- Hspec: a testing framework similar to RSpec with support for QuickCheck and HUnit.
- The Haskell Test Framework, HTF: integrates both Hunit and QuickCheck.

Haskell landscape

Ancillary Tools

- darcs: revision control system.
- haddock: documentation system.
- cabal: build system.
- stack: build system.
- hoogle: type-aware API search engine.

Haskell landscape

Static Analysis Tools

- hlint: detect common style mistakes and redundant parts of syntax, improving code quality.
- Sourcegraph: Haskell visualizer.

Dynamic Analysis Tools

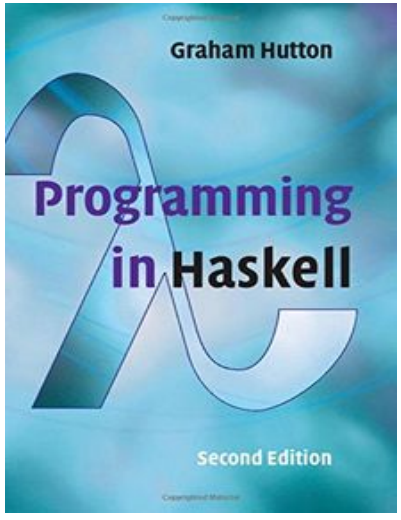
- criterion: powerful benchmarking framework.
- hpc: check evaluation coverage of a haskell program, useful for determining test coverage.

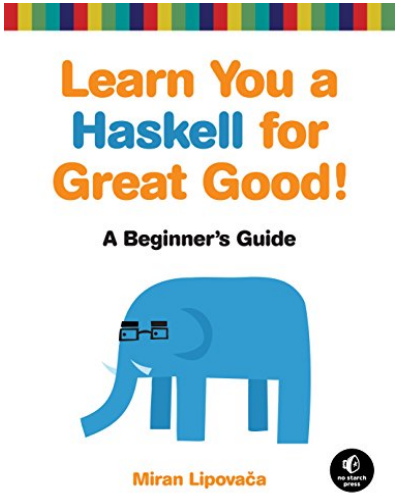
Haskell landscape

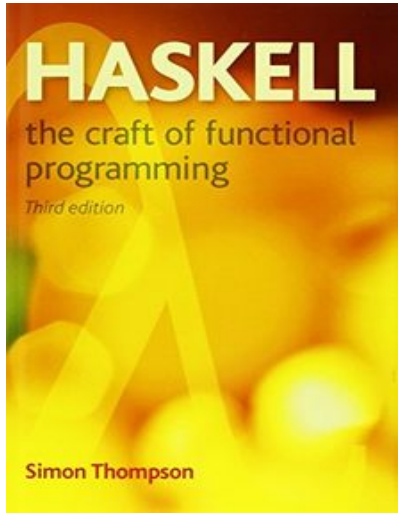
IDEs

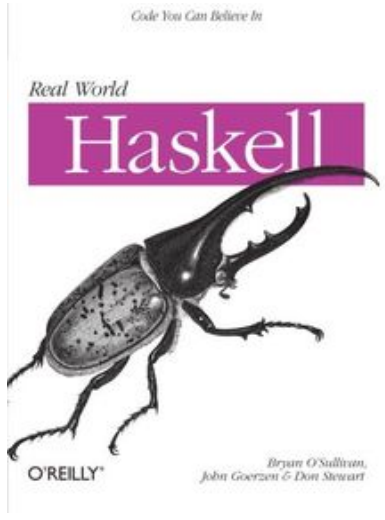
- VSCodium.
- IntelliJ.
- Vim.
- GNU Emacs.
- Haskell for Mac (commercial).
- Sublime Text (commercial)

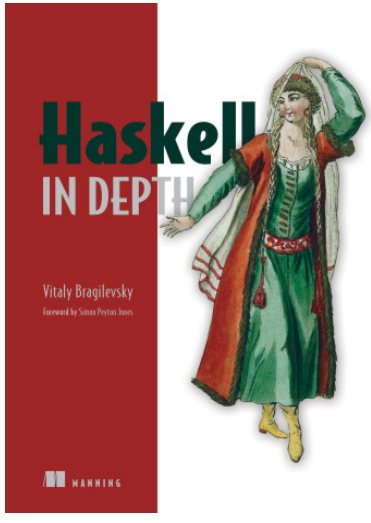
Haskell books

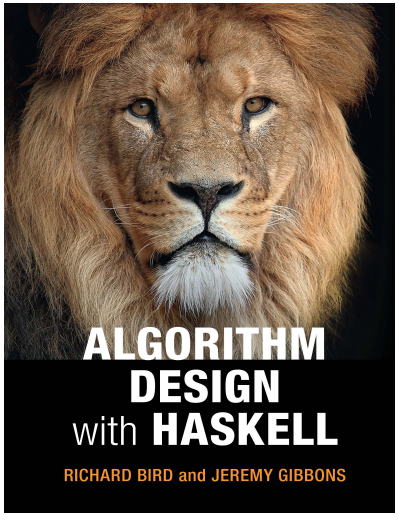


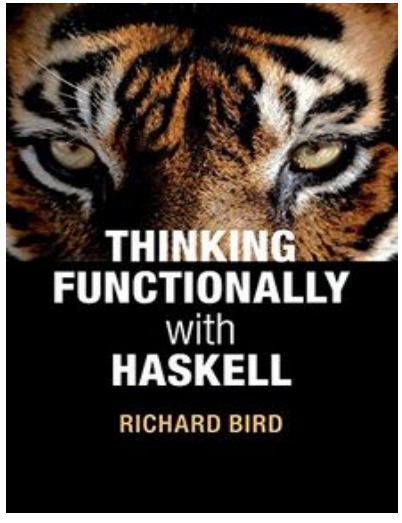


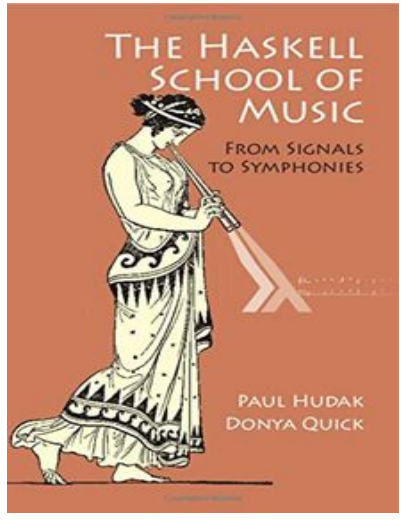




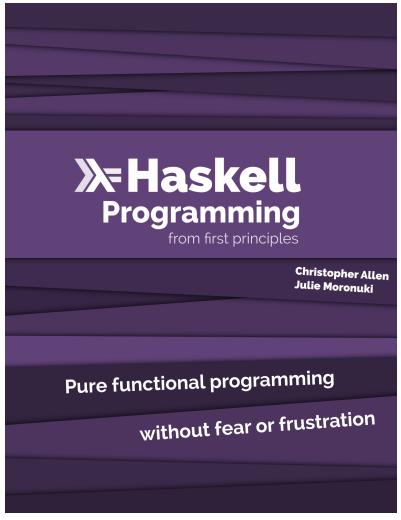


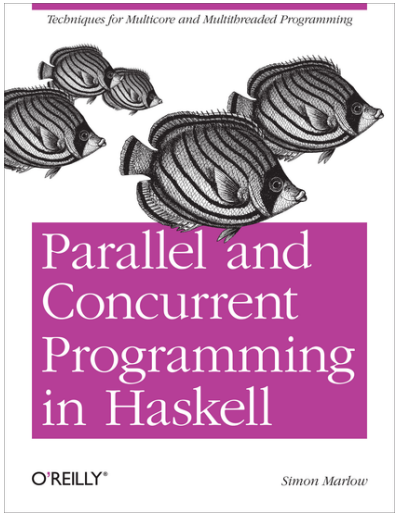


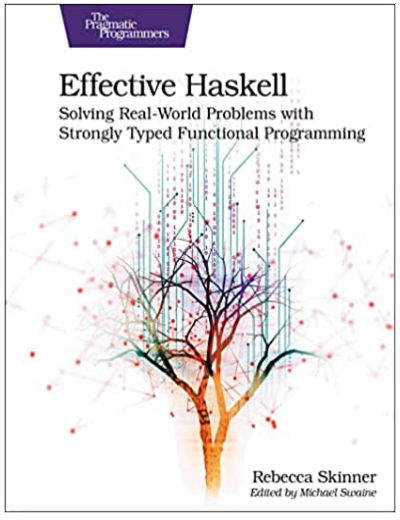


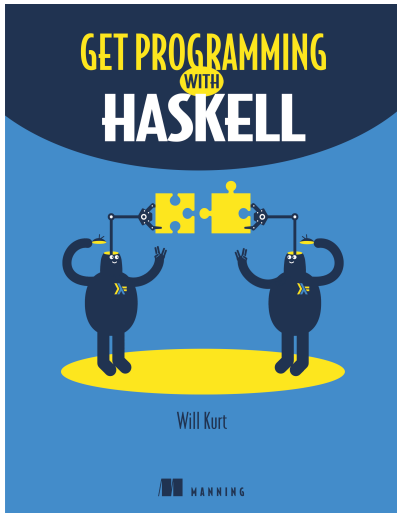


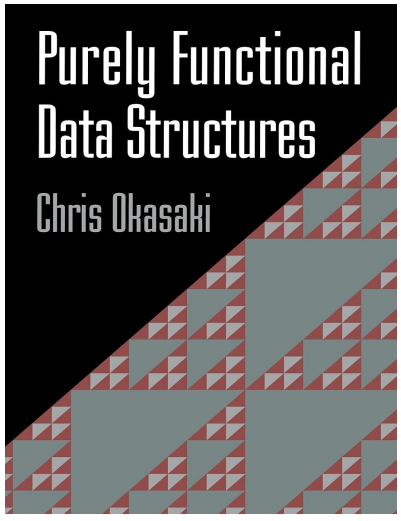
Haskell books

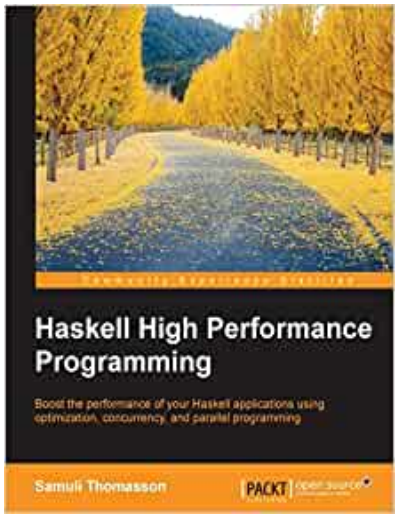


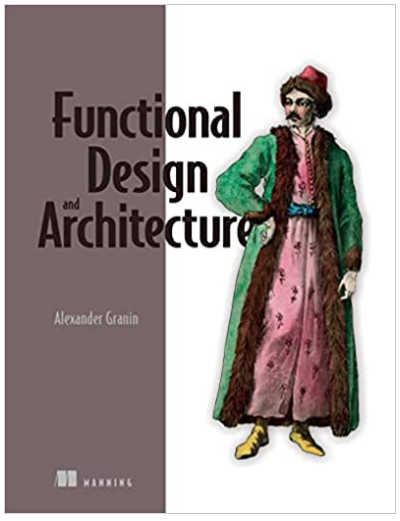


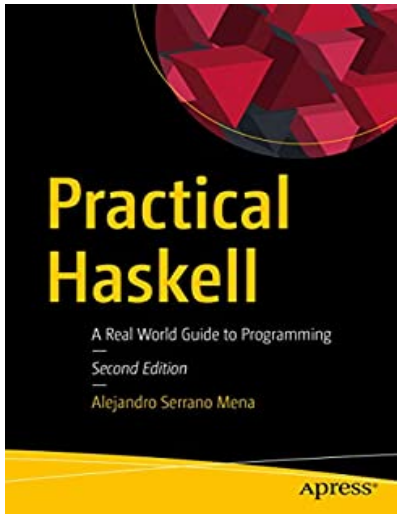




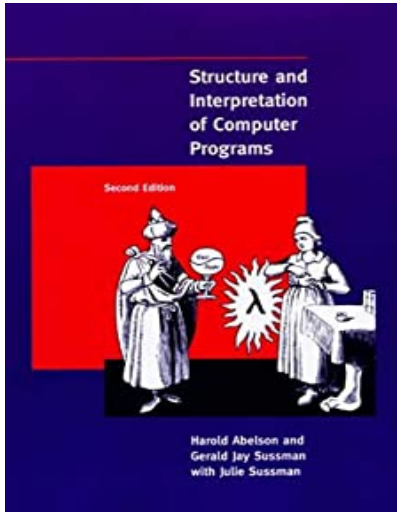


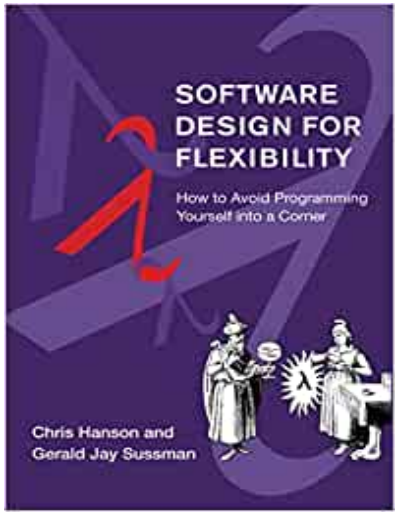


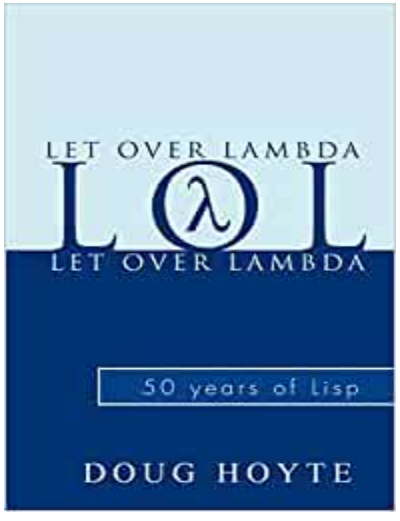


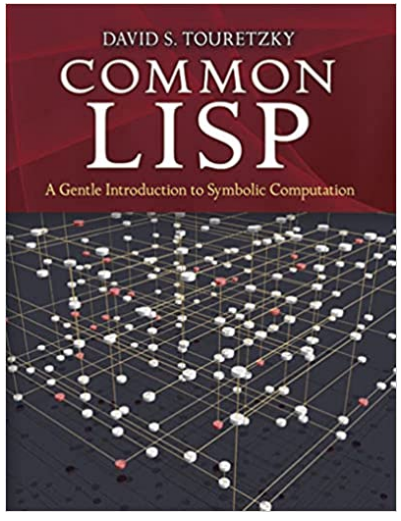


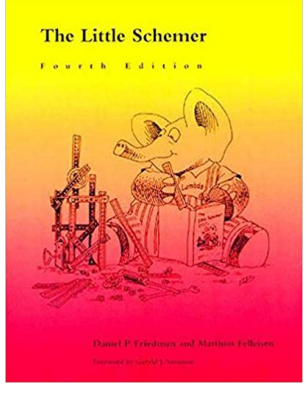
Functional programming books

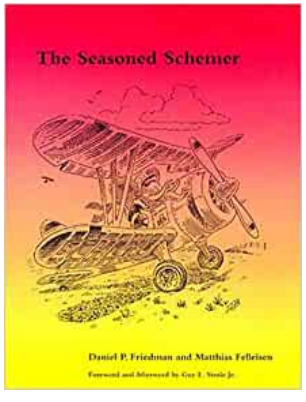


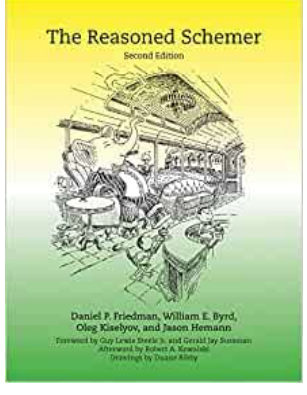


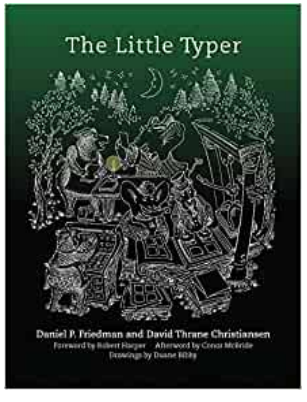


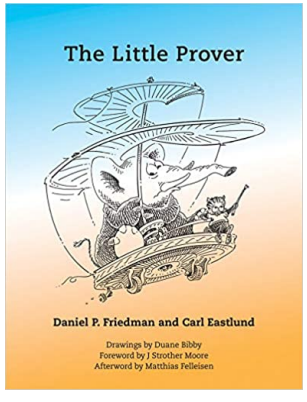












A taste of haskell

```haskell
sum :: Num a => [a] -> a
sum []       = 0
sum (x : xs) = x + sum xs
```

A taste of haskell

```
sum :: Num a => [a] -> a
sum []       = 0
sum (x : xs) = x + sum xs
```

```
  sum [1,2,3]
=   { applying function sum }
  1 + sum [2,3]
=   { applying function sum }
  1 + 2 + sum [3]
=   { applying function sum }
  1 + 2 + 3 + sum []
=   { applying function sum }
  1 + 2 + 3 + 0
=   { applying function +   }
  3 + 3 + 0
=   { applying function +   }
  6 + 0
=   { applying function +   }
  6
```

A taste of haskell

```haskell
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x : xs) = qsort smaller ++ [x] ++ qsort larger
  where
    smaller = [x' | x' <- xs, x' <= x]
    larger  = [x' | x' <- xs, x' >  x]
```

A taste of haskell

```haskell
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x : xs) = qsort smaller ++ [x] ++ qsort larger
  where
    smaller = [x' | x' <- xs, x' <= x]
    larger  = [x' | x' <- xs, x' >  x]
```

```
  qsort [x]
=   { applying function qsort }
  qsort [] ++ [x] ++ qsort [x]
=   { applying function qsort }
  [] ++ [x] ++ []
=   { applying function ++ (twice) }
  [x]
```

A taste of haskell

```haskell
qsort :: Ord a => [a] -> [a]
qsort [] = []
qsort (x : xs) = qsort smaller ++ [x] ++ qsort larger
  where
    smaller = [x' | x' <- xs, x' <= x]
    larger  = [x' | x' <- xs, x' >  x]
```

```
  qsort [3,5,1,4,2]
=   { applying function qsort }
  qsort [1,2] ++ [3] ++ qsort [5,4]
=   { applying function qsort (twice) }
  (qsort [] ++ [1] ++ qsort [2]) ++ [3] ++ (qsort [4] ++ [5] ++ qsort [])
=   { applying function qsort (four times) }
  ([] ++ [1] ++ [2]) ++ [3] ++ ([4] ++ [5] ++ [])
=   { applying function ++ (four times) }
  [1,2] ++ [3] ++ [4,5]
=   { applying function ++ (twice) }
  [1,2,3,4,5]
```

First steps

Glasgow Haskell Compiler

- The *Glasgow Haskell Compiler* (GHC) is the state-of-the-art open source implementation of Haskell
- The GHC if freely available for a range of operating systems from the Haskell home page http://www.haskell.org
- We recommand downloading the *Haskell Platform*
- Once installed, the interface GHCi system can be started from the terminal command prompt by simply typing `ghci`.

GHCi

```
λ> 1+2+3
6

λ> 1+2*3
7

λ> (1+2)*3
9

λ> 2-3+4
3

λ> 2*3/4
1.5
```

GHCi

```
λ> 2*pi
6.283185307179586

λ> (1 + sqrt 5) / 2
1.618033988749895

λ> log 2
0.6931471805599453
```

GHCi

```
λ> 2^3^4
2417851639229258349412352

λ> (2^3)^4
4096

λ> ceiling 2.6
3

λ> floor 2.6
2

λ> round 2.6
3

λ> (sin pi)^2 + (cos pi)^2
1.0
```

GHCi

```
λ> x = 42
λ> x+1
43

λ> let x = 42 in x+1
43

λ> let x = 1 in let x = 2 in x
2

λ> x = 1
λ> x = x+1
λ> x
^CInterrupted.

λ> y = y+1
λ> y
^CInterrupted.
```

GHCi

```
λ> "Haskell rocks!"
"Haskell rocks!"

λ> "Haskell " ++ "rocks!"
"Haskell rocks!"

λ> "Haskell " <> "rocks!"
"Haskell rocks!"

λ> ['H','a','s','k','e','l','l',' ','r','o','c','k','s','!']
"Haskell rocks!"
```

GHCi

Command	Meaning
`:load` *name*	load script *name*
`:reload`	reload current script
`:set editor` *name*	set editor to *name*
`:edit` *name*	edit script *name*
`:edit`	edit current script
`:type` *expr*	show type of *expr*
`:?`	show all commands
`:quit`	quit GHCi
...	

GHCi

```
λ> :type 1
1 :: Num a => a

λ> :type 2.5
2.5 :: Fractional a => a

λ> :type 5/2
5/2 :: Fractional a => a

λ> :type 5 `div` 2
5 `div` 2 :: Integral a => a
```

GHCi

```
λ>  :type 1+2
1+2 :: Num a => a

λ>  :type (+)
(+) :: Num a => a -> a -> a

λ>  :type (1 +)
(1 +) :: Num a => a -> a

λ> :type (+ 1)
(+ 1) :: Num a => a -> a
```

GHCi

```
λ>  :type 2.5
2.5 :: Fractional a => a

λ>  :type 5/2
5/2 :: Fractional a => a

λ>  :type (/)
(/) :: Fractional a => a -> a -> a

λ>  :type (/ 2)
(/ 2) :: Fractional a => a -> a
```

GHCi

```
λ> :type pi
pi :: Floating a => a

λ> :type sqrt 2
sqrt 2 :: Floating a => a

λ> :type cos
cos :: Floating a => a -> a
```

GHCi

```
λ > fact n = if n == 0 then 1 else n * fact (n-1)

λ > :type fact
fact :: (Eq a, Num a) => a -> a

λ > fact 5
120

λ > fact 0
1

λ > fact 5.0
120.0

λ > fact 2.5
^CInterrupted.
```

GHCi

```
λ> f = fact

λ> :type fact
fact :: (Eq a, Num a) => a -> a

λ> f 5
120

λ> f (f 3)
720
```

GHCi

```
λ> 'a'
'a'

λ> :type 'a'
'a' :: Char

λ> 'abc'
error: Syntax error on 'abc'

λ> 'a':"bc"
"abc"
```

GHCi

```
λ> "abc"
"abc"

λ> :type "abc"
"abc" :: String

λ> "abc" ++ "def"
"abcdef"

λ> :type (++)
(++) :: [a] -> [a] -> [a]
```

Types and classes

Basic concepts

- In Haskell every expression must have a type.
- A type is a collection of related values.
- We use the notation `v :: T` to mean that `v` is a value in the type `T`.

Example

```
True  :: Bool
False :: Bool
not   :: Bool -> Bool
(&&)  :: Bool -> Bool -> Bool
(||)  :: Bool -> Bool -> Bool
```

Basic types

- `Bool` - Logical values.
- `Char` - Single characters.
- `String` - Strings of characters.
- `Int` - Fixed-precision integers.
- `Integers` - Arbitrary-precision integers.
- `Float` - Since-precision floating-point numbers.
- `Double` - Double-precision floating-point numbers.

List types

- A list is a sequence of elements of the same type, with the elements being enclosed in square parentheses and separated by commas.
- We write `[T]` for the type of all lists whose elements have type `T`.
- The number of elements in a list is called its length.
- The list `[]` of length zero is called the empty list.
- `[]` and `[[]]` (and `[[[]]]`, `[[[[]]]]`, ...) are different lists.

List types

```
λ> :type []
[] :: [a]

λ> :type [1,2,3,4,5]
[1,2,3,4,5] :: Num a => [a]

λ> :type ['a', 'b', 'c', 'd']
['a', 'b', 'c', 'd'] :: [Char]

λ> :type ["ab", "cd", "ef", "gh"]
["ab", "cd", "ef", "gh"] :: [String]

λ> :type "ab" == :type "cd"
error: parse error on input ':'
```

List types

```
λ> :type [cos, sin]
[cos, sin] :: Floating a => [a -> a]

λ> :type [1, 'a']
error: No instance for (Num Char) arising from the literal '1'

λ> :type [[1],[2,3],[4,5,6]]
[[1],[2,3],[4,5,6]] :: Num a => [[a]]

λ> :type [[[1]],[[2,3],[4,5,6]]]
[[[1]],[[2,3],[4,5,6]]] :: Num a => [[[a]]]
```

Tuple types

- A tuple is a sequence of components of possibly different types, with the components being enclosed in round parentheses and separated by commas.
- We write `(T1, T2, ..., Tn)` for the type of all tuples whose i-th component have type `Ti` for any $1 \leqslant i \leqslant n$.
- The number of elements in a tuple is called its arity.
- The tuple `()` of arity zero is called the empty tuple.
- Tuple of arity one are not permitted.

Tuple types

```
λ> :type ()
() :: ()

λ> :type (1,'a')
(1,'a') :: Num a => (a, Char)

λ> :type (1,2,'a',"abc")
(1,2,'a',"abc") :: (Num a, Num b) => (a, b, Char, String)

λ> :type (sqrt, 'a')
(sqrt, 'a') :: Floating a => (a -> a, Char)

λ> :type (1, ('a', "cd"))
(1, ('a', "cd")) :: Num a => (a, (Char, String))
```

Tuple types

```
λ> :type (1, ('a', "cd"))
(1, ('a', "cd")) :: Num a => (a, (Char, String))

λ> :type (1, [cos, sin])
(1, [cos, sin]) :: (Floating a1, Num a2) => (a2, [a1 -> a1])

λ> :type (1)
(1) :: Num a => a

λ> let t = (1,2) in (t, 3)
((1,2),3)

λ> let t = (1,t)
error: Couldn't match expected type 'b' with actual type '(a, b)'
```

Function types

- A function is a mapping of one type to results of another type.
- We write `T1 -> T2` for the type of all functions that map arguments of type `T1` to results of type `T2`.
- There is no restriction that function must be total on their argument type.

Function types

```
λ> :type not
not :: Bool -> Bool

λ> :type even -- :type odd
even :: Integral a => a -> Bool

λ> :type mod
mod :: Integral a => a -> a -> a

λ> add x y = x+y
λ> :type add
add :: Num a => a -> a -> a

λ> add' (x,y) = x+y
λ> :type add'
add' :: Num a => (a, a) -> a
```

Curried functions

- Currying is the process of transforming a function that takes multiple arguments in a tuple as its argument, into a function that takes just a single argument and returns another function which accepts further arguments, one by one, that the original function would receive in the rest of that tuple.
- The function arrow `->` in type is assumed to associate to the right.

  The type

  ```
  T1 -> T2 -> T3 -> ... -> Tn
  ```

  means

  ```
  T1 -> (T2 -> (T3 -> ( ... -> Tn)...))
  ```

Curried functions

The type

```
a1 -> a2 -> a3
```

means

```
a1 -> (a2 -> a3)
```

Curried functions

The type

```
a1 -> a2 -> a3 -> a4
```

means

```
a1 -> (a2 -> (a3 -> a4))
```

Curried functions

The type

```
a1 -> a2 -> a3 -> a4 -> a5
```

means

```
a1 -> (a2 -> (a3 -> (a4 -> a5)))
```

Curried functions

Multiplying three integers

```
--  mult :: Int -> (Int -> (Int -> Int))
mult :: Int -> Int -> Int -> Int
mult x y z = x*y*z
```

Curried functions

Multiplying three integers

```
--  mult :: Int -> (Int -> (Int -> Int))
mult :: Int -> Int -> Int -> Int
mult x y z = x*y*z

λ> mult 2 3 4
24

λ> :type mult 2
mult 2 :: Int -> Int -> Int

λ> :type mult 2 3
mult 2 3 :: Int -> Int

λ> :type mult 2 3 4
mult 2 3 4 :: Int
```

Curried functions

Multiplying three integers

```
--  mult :: Int -> (Int -> (Int -> Int))
mult :: Int -> Int -> Int -> Int
mult x y z = x*y*z

λ> mult2 = mult  2
λ> mult3 = mult2 3
λ> mult3 4
24

λ> :type mult2
mult2 :: Int -> Int -> Int

λ> :type mult3
mult3 :: Int -> Int
```

Polymorphic types

- Parametric polymorphism refers to when the type of a value contains one or more (unconstrained) type variables, so that the value may adopt any type that results from substituting those variables with concrete types.
- For example, the function `id :: a -> a` contains an unconstrained type variable `a` in its type, and so can be used in a context requiring `Char -> Char` or `Integer -> Integer` or `(Bool -> Bool) -> (Bool -> Bool)` or any of a literally infinite list of other possibilities.
- The empty list `[] :: [a]` belongs to every list type.

Polymorphic types

```
λ > length []
0

λ > length [1,3,5,7,2,4,6,8]
8

λ > length ["Huey","Dewey","Louie"]
3

λ > length [sin, cos, tan]
3
```

Polymorphic types

```
λ >  :type length
length :: Foldable t => t a -> Int

λ >  :info length
type Foldable :: (* -> *) -> Constraint
class Foldable t where
  length :: t a -> Int
  ...
  -- Defined in 'Data.Foldable'
```

Overloaded types

- A type that contains one or more class constraints is called overloaded.
- Class constraints are written in the form `C a`, where `C` is the name of the class and `a` is a type variable.

Overloaded types

```
λ> 1 + 2
3

λ> :type 1
1 :: Num a => a

λ> :type 1 + 2
1 + 2 :: Num a => a
```

```
λ> 1.0 + 2.0
3.0

λ> :type 1.0
1.0 :: Fractional a => a

λ> :type 1.0 + 2.0
1.0 + 2.0 :: Fractional a => a
```

```
λ> sqrt 2 + sqrt 3
3.1462643699419726

λ> :type sqrt 2
sqrt 2 :: Floating a => a

λ> :type sqrt 2 + sqrt 3
sqrt 2 + sqrt 3 :: Floating a => a
```

Overloaded types

```
λ> :type (+)
(+) :: Num a => a -> a -> a

λ> :type (-)
(-) :: Num a => a -> a -> a

λ> :type (*)
(*) :: Num a => a -> a -> a

λ> :type (/)
(/) :: Fractional a => a -> a -> a

λ> :type sqrt
sqrt :: Floating a => a -> a
```

Basic classes

- A class is collection of types that support certain overloaded operations called methods.
- Haskell provides a number of basic classes that are built-in to the language.

Basic classes

`Eq` – Equality types

This class contains types whose values can be compared for equality and inequality using the following two methods:

```
(==) :: a -> a -> Bool
(/=) :: a -> a -> Bool
```

All the basic types `Bool`, `Char`, `String`, `Int`, `Integers`, `Float` and `Double` are instances of the `Eq` class.

Eq – Equality types

```
λ> True == True
True

λ> 'a' == 'b'
False

λ> "abc" == "abc"
True

λ> 2.5 == 5.2
False
```

Eq – Equality types

```
λ> ('a', 1) == ('b', 1)
False

λ> (1, 2, 3) == (1, 2)
error: Couldn't match expected type: (a0, b0, c0) with actual
       type: (a1, b1)

λ> [1,2,3] == [1,2,3,4]
False

λ> cos == cos
error: No instance for (Eq (Double -> Double)) arising from a
       use of '=='
```

`Ord` – Ordered types

This class contains types that are instances of the equality class `Eq`, but in addition these values are totally ordered, and as such can be compared using the following six methods:

```
(<)  :: a -> a -> Bool
(<=) :: a -> a -> Bool
(>)  :: a -> a -> Bool
(>=) :: a -> a -> Bool
min  :: a -> a -> a
max  :: a -> a -> a
```

All the basic types `Bool`, `Char`, `String`, `Int`, `Integers`, `Float` and `Double` are instances of the `Ord` class.

`Ord` – Ordered types

```
λ> False < True
True

λ> "elegant" < "elephant"
True

λ> "a" < "ab"
True

λ> 'b' > 'a'
True

λ> [1,2,3] <= [1,2]
False

λ> [] < [1]
True
```

Basic classes

`Ord` – Ordered types

```
λ> (1,2) < (1,3)
True

λ> (1,2,3) < (1,1)
error: Couldn't match expected type: (a0, b0, c0) with actual
       type: (a1, b1)

λ> [True] < [False,False]
False

λ> (False,False) <= (False,True)
True
```

Basic classes

`Ord` – Ordered types

```
λ >
```

```
λ > min ('a',2) ('a',1)
('a',1)
```

```
λ > max ('a',2) ('a',1)
('a',2)
```

```
λ > sin < cos
error: No instance for (Ord (Double -> Double)) arising from a
       use of '<'
```

```
λ > (1, sin) > (2, cos)
error: No instance for (Ord (Double -> Double)) arising from a
       use of '>'
```

Basic classes

`Show` – **Showable types**

This class contains types that can be converted into strings of characters using the following method:

```
show :: a -> String
```

All the basic types `Bool`, `Char`, `String`, `Int`, `Integers`, `Float` and `Double` are instances of the `Show` class.

Show – Showable types

```
λ> show True
"True"

λ> show 'a'
"'a'"

λ> show "abc"
"\"abc\""

λ> show [1,2,3]
"[1,2,3]"

λ> show (1, True, [1,2,3])
"(1,True,[1,2,3])"
```

`Read` – **Readable types**

This class is dual to `Read` and contains types whose values can be converted from string of characters using the following method:

```
read :: String -> a
```

All the basic types `Bool`, `Char`, `String`, `Int`, `Integers`, `Float` and `Double` are instances of the `Read` class.

Read – Readable types

```
λ> read "False" :: Bool
False

λ> read "'a'" :: Char
'a'

λ> read "\"abc\"" :: String
"abc"

λ> read "[1,2,3]" :: [Int]
[1,2,3]

λ> read "(1, True, [1,2,3])" :: (Int, Bool, [Int])
(1,True,[1,2,3])
```

`Num` – Numeric types

This class contains types whose values are numeric, and as such can be processed using the following six methods:

```
(+) :: a -> a -> a
(-) :: a -> a -> a
(*) :: a -> a -> a
negate :: a -> a
abs    :: a -> a
signum :: a -> a
```

Note that the `Num` class does not provide a division method.

Basic classes

`Num` – Numeric types

```
λ> 1+2
3

λ> 1-2
-1

λ> 1.0+2.0
3.0

λ> 2*3
6

λ> 2.0*3.0
6.0
```

`Num` – Numeric types

```
λ> negate 3.0
-3.0

λ> negate (-2)
2

λ> abs(-1.5)
1.5

λ> signum 3
1

λ> signum (-3)
-1
```

`Integral` – **Integral types**

This class contains types that are instances of the numeric class `Num`, but in addition whose values are integers, and as such support the method of integer division and integer remainder:

```
div :: a -> a -> a
mod :: a -> a -> a
```

Basic classes

`Integral` – Integral types

```
λ> div 7 2
3

λ> 7 `div` 2
3

λ> 8 `div` 2
4

λ> 7 `mod` 2
1

λ> 8 `mod` 2
0
```

Basic classes

`Integral` – Integral types

```
λ> (-7) `div` 2
-4

λ> (-7) `div` (-2)
3

λ> (-7) `mod` 2
1

λ> (-7) `mod` (-2)
-1
```

`Fractional` – **Fractional types**

This class contains types that are instances of the numeric class `Num`, but in addition whose values are non-integral, and as such support the method of integer fractional division and fractional reciprocation:

```
(/) :: a -> a -> a
recip :: a -> a -> a
```

The basic types `Float` and `Double` are instances of the `Fractional` class.

`Fractional` – Fractional types

```
λ> 7.0 / 2.0
3.5

λ> 2.0 / 7.0
0.2857142857142857

λ> recip 2.0
0.5

λ> recip 1.0
1.0
```